Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

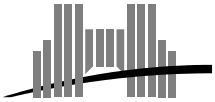
A Repair Mechanism for Fault-Tolerance for Tree-Structured Peer-to-Peer Systems

Eddy Caron ,
Frédéric Desprez ,
Charles Fourdrignier ,
Franck Petit,
Cédric Tedeschi

Oct 2006

Research Report N° 2006-34

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



A Repair Mechanism for Fault-Tolerance for Tree-Structured Peer-to-Peer Systems

Eddy Caron , Frédéric Desprez , Charles Fourdrignier , Franck Petit, Cédric Tedeschi

Oct 2006

Abstract

Facing the limits of traditional tools of resource management within computational grids (related to scale, dynamicity, etc. of the platforms newly considered), new approaches, based on peer-to-peer technologies are emerging. The resource discovery and in particular the service discovery is concerned by this evolution. Among the solutions, a promising one is the indexing of resources using trie structures and more particularly prefix trees. The major advantages of trie-structured approaches is the capability to support search queries on ranges of values with a latency growing logarithmically in the number of nodes in the trie. Those techniques are easy to extend to multicriteria searches. One drawback of using tries is its inherent poor robustness in a dynamic environment, where nodes join and leave the network, leading to the split of the tree into a forest, which results in the impossibility to route requests. Within most recent approaches, the fault-tolerance is a prevention mechanism, often replication-based. The replication can be costly in term of resources required. In this paper, we propose a fault-tolerance protocol that reconnects subtrees *a posteriori*, after crashes, to have again a connected graph and then reorder the nodes to rebuild a consistent tree.

Keywords: Fault tolerance, peer-to-peer, prefix trees

Résumé

Face aux limites des outils traditionnels de gestion de ressources dans les grilles de calcul (mauvais passage à l'échelle, non prise en compte de la dynamicité du réseau, etc.), des alternatives fondées sur les technologies pair-à-pair sont en train d'émerger. La découverte de ressources et en particulier des services de calcul est touchée par cette évolution. Parmi ces solutions, il existe des approches prometteuses fondées sur des arbres *lexicographiques*. L'intérêt de telles approches repose sur la possibilité d'effectuer des requêtes sur des intervalles de valeurs ainsi que la possibilité de réaliser de l'autocomplétion sur les chaînes de recherche en temps logarithmique en la taille de l'arbre. Ces techniques s'étendent facilement à des recherches multicritères. Cependant la structure en arbre est fragile et peut éclater en une forêt si l'un des nœuds vient à quitter le réseau, rendant ainsi impossible le routage de certaines requêtes et n'offrant au client qu'une vue partielle des services. Dans la plupart de ces approches, la tolérance aux pannes, indispensable dans les environnements dynamiques à large échelle, est préventive (réalisée *a priori*) et se fonde sur la réplication, qui est coûteuse en termes de ressources et de temps. Dans ce papier, nous présentons un protocole tolérant aux pannes de nœuds, complémentaire à la réplication, dans les arbres lexicographiques. Il se fonde sur la reconnexion et la réparation *a posteriori* d'arbres qui ont subi la perte d'un ou plusieurs nœuds.

Mots-clés: Tolérance aux pannes, pair-à-pair, arbres de préfixes

1 Introduction

These last few years have seen the development of large scale grids connecting distributed resources (computation resources, storage facilities, computation libraries, etc.) in a seamless way. This is now an efficient alternative to supercomputers to solve large problems such as high energy physics, simulation, bioinformatic, etc. However, existing middlewares used in grids require most of the time a stable and centralized infrastructure. They usually loose their performance on dynamic and large scale platforms without centralized management of resources. To cope with the characteristics of these emerging kind of platforms, it has been suggested to use peer-to-peer technologies within computational grids [8].

Peer-to-peer technologies offer algorithms allowing the search and retrieval of objects over the net (data items, files, services, etc.). Among these technologies, Distributed Hash Tables (DHT) were initially designed for very large scale platforms, for example to share files over the Internet. However, DHTs have several major drawbacks. Among them, their discovery mechanism usually works on exact searches of a given key. Some work has then been done to allow complex requests to be submitted over DHTs or more generally in **structured** peer-to-peer systems, i.e. systems based on request routing. Some of these works are based on *tries* (also called *prefix trees*). A trie structure supports range queries in a logarithmic time in the number of nodes of the trie.

Fault-tolerance is a mandatory feature for peer-to-peer systems to avoid the loss of data stored on nodes and to allow a correct routing of messages. The crash of one or several nodes in a trie leads to the loss of objects references stored in the trie and to the split of the trie into several subtries, also called a *forest*. Fault-tolerance within structured peer-to-peer systems usually uses replication. Using such an approach, each node and each link of the trie would have to be duplicated k times, k being the replication factor. Keeping such structure up is costly, mainly in terms of resources used. Afterward, the purpose is to find for the value of k the right trade-off between the replication cost and the robustness of the system. In this paper, we study an alternative to the replication approach based on the reconnection of the subtries and the *a posteriori* reordering of a consistent trie. When the trie is disconnected, a first solution consists in rebuilding a trie adding nodes of remaining subtries one by one. This naive method can lead to a prohibitive cost when the number of remaining nodes is large (which is usually the case in peer-to-peer systems). For example, loosing one node can lead to a complete reconstruction of the trie. A second approach consists in reconnecting the subtries to get the original trie back at a minimum cost. This is this kind of algorithm we describe in this paper in a distributed and asynchronous environment. It can also be used to complete the replication process.

A brief history of peer-to-peer technologies is provided in Section 2, followed by the formal description of the particular trie structure we use (Section 3) and of the distributed system we place ourselves. We focus our study on fault-tolerance mecanisms related to them. Then, in Section 4 we present the repair algorithm we designed and give its proof before a conclusion and future work Section.

2 Related Work

With the spread of the peer-to-peer technologies going along with the file sharing over the Internet, purely decentralized search systems have emerged. Such tools first took the shape

of *unstructured* mechanisms, *i.e.,* based on the flooding of search requests [10, 9]. These mechanisms resulted in overloading the network while providing non-exhaustive responses. Addressing both the scalability and the exhaustiveness issues within peer-to-peer systems, the distributed hash tables [13, 14, 18, 20], *a.k.a.,* the *structured* peer-to-peer group, are highly scalable in the sense that the number of logical hops required to route and the local state grows logarithmically with the number of nodes participating in the system. Moreover, DHTs prevent from loosing routing paths and objects' references by use of replication and periodic scans. Unfortunately, DHTs present several major drawbacks (homogeneous capacity assumptions, topology awareness, etc.). Among them, the rigidity of the requesting mechanism, *i.e.,* exact match on a given key hinders its use over real search systems.

A series of work gives the opportunity to allow flexible meanings of retrieval over structured peer-to-peer networks. First achievement in this way has been the ability to describe resources with semi-structured language, such XML, as described in [3]. [19] enhances DHTs with traditional database operations. Several approaches, based on space filling curves, such as Squid [15] or [17] support multi-dimensional range queries. [1] maps one-dimensional data space to d-dimensional Cartesian space by using the inverse Hilbert mapping. Built on top of multiple DHTs, SWORD [11] is an information service aiming at discovering computing resources on the grid by answering multi-attribute range queries.

We focus in this work on trie-structured retrieval solutions, also supporting range queries but outperforming previous approaches in the sense that logarithmic (or constant if we assume an upper bound on the depth of the trie) latency is achieved by parallelizing the resolution of the query in the several branches of the trie. Prefix Hash Tree (PHT) [12] builds a trie of the entire key-space on top of a DHT. The purpose of this architecture is to use the trie as a logical layer allowing complex searches on top of any DHT-like network. The architecture of PHT results in the multiplication of the complexities of the trie and of the underlying DHT.

The Skip Graphs structure proposed in [2] is similar to a trie but is built with the skip lists technology, allowing the use of their inherent fault-tolerance properties. But again, the complexity of the number of messages generated to process range queries is in $O(m \log(n))$, m being the number of nodes pertained by the range and n the total number of nodes in the graph.

Other approaches propose to rely on a trie for each purpose, *i.e.,* indexing the key-space, mapping the nodes of the trie on the network, and routing the requests. Among them, Nodewiz [4] assumes a set of static reliable nodes to host the trie, which is unfortunately hard to ensure on peer-to-peer platforms. P-Grid [7] builds a trie on the whole key-space (*i.e.,* the whole set of potential keys). Each leaf of this trie corresponds to a subset of the key-space. The fault-tolerance is achieved by probabilistic replication.

As a more general consideration, none of these approaches address the topology/physical locality awareness issue, *i.e.,* no information about the underlying network is taken into account to build the logical (overlay) network, what can raise a significant performance problem, physical locality being broken when the logical network is built. Moreover, the several fault-tolerance solutions are mostly replication-based, or DHT-based, also involving heavy replication mechanisms.

Initially designed for the purpose of service discovery over dynamic computational grids and attempting to solve the above drawbacks of existing approaches, we recently developed a novel architecture, based on a logical Greatest Common Prefix Tree formally described in Section 3, that is dynamically built as objects (services, but extensible to data items, files, etc.) are declared.

3 Preliminaries

Greatest Common Prefix Tree. Let an ordered alphabet A be a finite set of letters. Denote $\prec$ an order on A. A non empty *word* w over A is a finite sequence of letters $a_1, \ldots, a_i, \ldots, a_l$, $l > 0$. The *concatenation* of two words u and v, denoted $u \circ v$ or simply uv, is equal to the word $a_1, \ldots, a_i, \ldots, a_k, b_1, \ldots, b_j, \ldots, b_l$ such that $u = a_1, \ldots, a_i, \ldots, a_k$ and $v = b_1, \ldots, b_j, \ldots, b_l$. Let ϵ be the *empty word* such that for every word w, $w\epsilon = \epsilon w = w$. The *length* of a word w, denoted by $|w|$, is equal to the number of letters of w—$|\epsilon| = 0$.

A word u is a *prefix* (respectively, *proper prefix*) of a word v if there exists a word w such that $v = uw$ (resp., $v = uw$ and $u \neq v$). The *Greatest Common Prefix* (resp., *Proper Greatest Common Prefix*) of a collection of words $w_1, w_2, \ldots, w_i, \ldots$ ($i \geq 2$), denoted $GCP(w_1, w_2, \ldots, w_i, \ldots)$ (resp. $PGCP(w_1, w_2, \ldots, w_i, \ldots)$), is the longest prefix u shared by all of them (resp., such that $\forall i \geq 1, u \neq w_i$). A [*Proper*] *Greatest Common Prefix Tree* ([P]GCP Tree, also a particular kind of *trie*) is a labeled rooted tree such that both following properties are true for every node of the tree:

1. The node label is a proper prefix of any label in its subtree;
2. The node label is the Proper Greatest Common Prefix of all its son labels.

In the following we use the word **trie** to designate our PGCP tree.

Distributed Lexicographic Placement Table. The *distributed system* considered in this paper consists of a set of asynchronous physical nodes organized in a *Distributed Hash Tables (DHT)*. Each physical node maintains one or more nodes of the logical PGCP Tree. Note that a DHT is used, but it can be replaced by any system, distributed or not, allowing the retrieval of any node from any other node. We also consider that the potential existing fault-tolerance mechanisms provided by this layer are not used within our architecture. We propose in this paper a fault-tolerance mechanism at the PGCP Tree level.

When one wants to insert an object labeled o into the trie, a message is generated containing o, according to which the message is routed within the trie until reaching the node labeled v such that v is the smallest label in the trie that shares with o the greatest common prefix of any node of the trie with o. More formally, if L denotes the whole set of label currently in the trie, the set $U = \{l \in L \mid GCP(l, o) = p\}$ where $p = max_{|m|}\{m = PGCP(l, o), l \in L)$. The label of the target node is $t = min_{|w|}\{u \in U \mid u = pw\}$. Once found, the target node performs the insertion. If $t \neq o$, node(s) are created. If $o = tu$ ($u \neq \epsilon$), a new node labeled o is created as a new son of the node labeled t. If $t = ou$ ($u \neq \epsilon$), a new node is created as the father of the node labeled by t. Finally, if none of these conditions are satisfied, it means that o and t must be siblings but no node in the trie is labeled by their common prefix. Thus two nodes are created, a node labeled $GCP(o, t)$, father of the node labeled by t and also father of the other newly created node labeled by o. The distributed routing algorithm (that also performs the creation and the mapping of nodes) requires a number of hops bounded by twice the depth of the trie [5].

Physical nodes communicate by *message passing*. We assume two sending functions. The former, simply referred to *SEND*, is used by any physical node to send a message to another node asynchronously, i.e., without waiting any acknowledgement. The latter, called *SYNC-SEND*, waits for an acknowledgement for each message sent. We assume that each physical node may crash. So, when a physical node crashes, one or more logical nodes are lost.

4 Protocol

In this section, we give a detailed explanation of how the protocol works. We divide the algorithm code in two parts. The former shows the first phase developed with our technique during which a unique trie is recovered without considering any lexicographic property. During the second phase, the trie is reorganized to eventually form a distributed greatest common prefix tree.

4.1 Trie Recovery

After a node p detects the loss of its father ($p.father$), it searches for a new father to link on. Making a traversal of the DHT, Node p collects in Variable PN all the addresses of each remaining physical node. Collecting the addresses in PN, p builds the set of logical nodes stored by the physical nodes in PN. Next, using a PIF (*Propagation of Information with Feedback*) Protocol [6, 16], p computes T, the set of logical nodes in its subtrie, which is made of its "real" descendants and its "temporary" relinked descendants. This first step of the recovery protocol ends when p chooses a temporary father ($p.tmpfather$) in the subset $N \setminus T$. When, a node q is linked to a node p, then p considers q as a temporary son—stored in $p.tmpsons$. Note that Variable $p.tmpsons$ is required to compute T using a PIF in the subtrie of p. If $N \setminus T = \emptyset$ (i.e., there is no node for which p may link on), then p is considered as the root of the trie.

The above technique suffers of a drawback: Several nodes without father may make which could become a "bad" choice. In particular, they can choose as a temporary father a node belonging to the subtrie of another node being in the same situation. By doing this in parallel, cycles may appear. Our strategy is to detect and to break *a posteriori* such cycles as follows.

After the choice of its temporary father tf, a node p sends a message "HELLO" with its ID ($p.id$) to tf. In the next step, tf transmits the message to its own father, and so on. Step by step, one of the two following situations eventually arises:

1. The "real" root of the trie receives the message "HELLO". In that case, the root notifies p that it is not involved in a cycle.

2. The message is received by a "false" root, i.e., a node having also lost its own father. the false root propagates the message to its temporary father.

Note that, in the above latter case, due to asynchrony of the network, it is possible that the false root receives the message "HELLO" sent by p before it executed its own recovery phase. In that case, the false root is still without a temporary father. The message "HELLO" is then delayed until the false root chooses its own temporary father.

Therefore, the message "HELLO" sent by p keeps circulating among its ancestors, carrying the list of false roots' IDs which were met during its traversal. Upon receipt of a message "HELLO", if the first item of the list carried by the message is equal to the ID of the receiver, then a cycle is detected. In that case, a leader election is computed among the IDs of the list—e.g., by choosing the smallest ID. The leader becomes the root of the subtrie, breaks its link which its father, and executes the recovery phase again. (The other "false" roots involved in the cycle remain connected to the subtrie rooted by the leader.) Note that a cycle may be created again. However, in the worst case, at each relaunching of the recovery phase, at

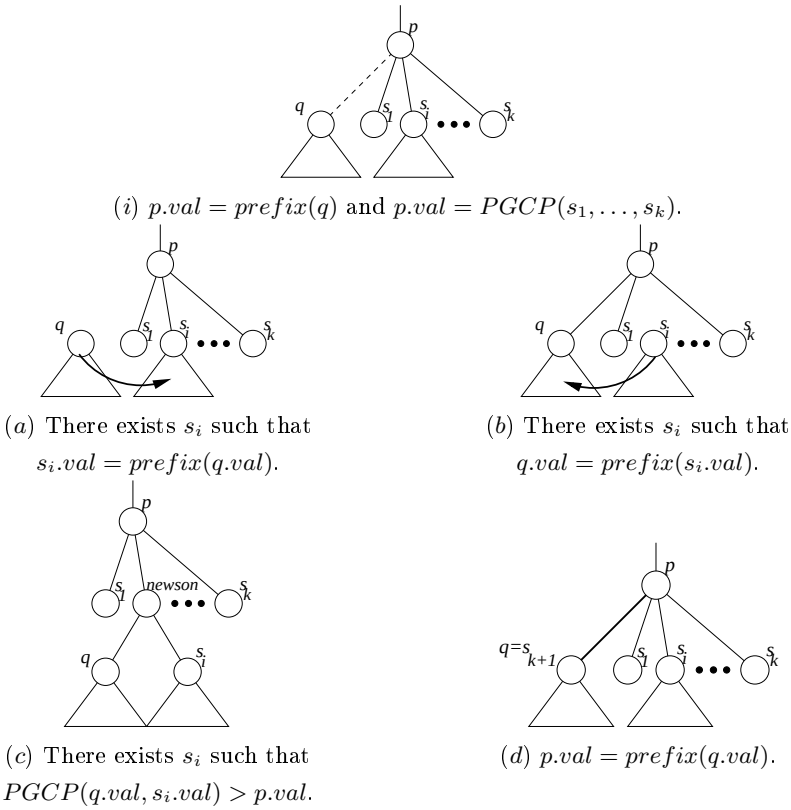
least one subtrie becomes the subtrie of one false root. In other words, the number of cycles is periodically divided by at least 2. Therefore, the system eventually contains one (rooted) trie only.

4.2 Trie Reorganization

The trie reorganization is initiated once the trie recovery is done. Each node p having a temporary son q—i.e., q is a false root with its subtrie—initiates a routing mechanism closed to the original key insertion [5]. Let us consider the following cases:

1. The value $p.val$ is a prefix of the value of q—Figure 1, Case (i). In that case, q (and its subtrie) is placed in the subtrie of p following one of the four cases shown in Figure 1, Cases (a) to (d).

2. The value $p.val$ is not a prefix of the value of q. Then, p moves q to its father which now has the responsibility to place q.



(i) $p.val = prefix(q)$ and $p.val = PGCP(s_1, \ldots, s_k)$.

(a) There exists s_i such that $s_i.val = prefix(q.val)$.

(b) There exists s_i such that $q.val = prefix(s_i.val)$.

(c) There exists s_i such that $PGCP(q.val, s_i.val) > p.val$.

(d) $p.val = prefix(q.val)$.

Figure 1: A false root q is linked to a node p such that $p.val = prefix(q.val)$.

Note that new services may keep inserting during the trie reconstruction. So, a new subtrie may have been created at the same place where the false root initially was. Thus, our method requires to take in account that any false root being placed in the trie can meet a node having the same value. In that case, the two tries must be merged. That is the aim of the merging protocol, initiated by the sending of a message "MERGE". Upon receipt of this message, a

node p executes Procedure *Gluing*(q), which moves the sons of q to p before withdrawing q from the trie (including the sons of q's father). Then, if necessary, p restarts recursively merging and placements among its sons, in order to merge both subtries eventually.

4.3 Correctness Proof

In this subsection, we discuss the correctness of our protocol. In order to do this, we first need to make the realistic assumption that under the considered context, the crash frequency is low enough to make the trie fully built sometime. (In the opposite way, the trie could never be built and unusable most of the time. More generally it is impossible to say anything about termination otherwise.) In other words, we fairly assume that no crash occurs after a crash until the trie is fully built, i.e., no two consecutive crashes interfere each other, at one given time.

Assuption 1 *If a node crashes at time t, then for every $t' > t$, no crash occurs.*

Lemma 2 *Under Assumption 1, the recovery protocol (Algorithm 1) terminates, and when this occurs, the system contains one trie only.*

Proof. The validation mainly consists in showing that the protocol terminates and that the reorganization of the trie is eventually initiated (by sending a message NOCYCLE).

Assume by contradiction that under Assumption 1, no node eventually sent a message NOCYCLE. So, neither Line 1.35 nor Line 1.37 in Algorithm 1 is executed. Note that in the first case (Line 1.35), the node becomes the "real" node after the crash of its father. So, in both cases, this means that NOCYCLE never reaches the "real" root of the trie. The height of the trie being finite, this means that every Message HELLO traverses cycles only. When a message HELLO is received by its initiator, the cycle is broken by the node which is elected among the false roots participating in the cycle—Lines 1.16 to 1.21. Therefore, cycles are created infinitely often. Let C be the number of created cycles. In the worst case, a cycle is made of at least two nodes. So, C is initially bounded by $F/2$, where F is the number of false root created by the crash. When a cycle is broken, at most one leader is elected. So, at most $C/2$ leaders are able to link another node again. In the next phase, the number of cycles is less than or equal to $C/2$. Since under Assumption 1, cycles may be created only when false roots are linked to other nodes (executing Lines 1.10 and 1.11), C never grows and is eventually equal to 0. This contradicts that cycles are created infinitely often. □

We now consider the phase of trie reorganization shown in Algorithm 2.

Lemma 3 *Under Assumption 1 and assuming that the system contains one trie only, the reorganization protocol (Algorithm 2) terminates, and when this occurs, the trie is a $PGCP$ tree.*

Proof. Clearly, each trie of the forest following the crash of a node is a $PGCP$ tree. So, its remains to show that executing Algorithm 2, the whole trie eventually satisfies the condition to be a $PGCP$ tree.

From the algorithm, it is easy to observe that, in the absence of merging, there are only two cases to consider depending on the value of Node p and its false son fs :

Algorithm 1 Recovery Protocol for each node p

```
1.01 upon receipt of <Disconnected from Father> do
1.02       PN := Physical Node Set in the DHT (collected by a DHT traversal);
1.03       N := Logical Node Set in PN (collected by polling the nodes in PN);
1.04       T := Logical Node Set in my subtrie (collected using a PIF wave)
1.05       using p.sons ∪ p.tmpsons;
1.06       if p.tmpfather ≠⊥ then send <DISCONNECT> to p.tmpfather;
1.07       if N \ T = ∅
1.08       then  //I am the root
1.09             p.father :=⊥; p.tmpfather :=⊥;
1.10       else  p.tmpfather := random choice among N \ T;
1.11             send-sync <LINK> to p.tmpfather;
1.12             send <HELLO,p.id> to p.tmpfather;
1.13       endif
1.14 upon receipt of <HELLO,list> from q do
1.15       if First(list) = p.id
1.16       then  //A cycle is detected
1.17             leader := LeaderElection(list);
1.18             if p = leader
1.19             then  Executes "upon receipt of <Disconnect from Father> do",
1.20                   except PN and N;
1.21             endif
1.22       elseif p.Father ≠⊥
1.23       then  send <HELLO,list> to p.father;
1.24       elseif p.tmpfather ≠⊥
1.25       then  list := list + p.id;
1.26             send <HELLO,list> to p.tmpfather
1.27       elseif p.father =⊥
1.28       then  // Both father and tmpfather are unknown, i.e.,
1.29             I am a false root which is still not linked
1.30             Executes "upon receipt of <Disconnect from Father> do"
1.31             if it is still not working;
1.32             if tmpfather ≠⊥
1.33             then  list := list + p.id;
1.34                   send <HELLO,list> to p.tmpfather;
1.35             else  send <NOCYCLE> to First(list);
1.36       else  // I am the real root, so there is no cycle.
1.37             send <NOCYCLE> to First(list);
1.38       endif
1.39 upon receipt of <NOCYCLE> from q do
1.40       send <MOVE,p> to p.tmpfather;
1.41       send-sync <UNLINK> to p.tmpfather;
1.42       p.tmpfather :=⊥;
1.43 upon receipt of <LINK> from q do
1.44       tmpsons := tmpsons ∪ {q};
1.45 upon receipt of <UNLINK> from q do
1.46       tmpsons := tmpsons \ {q};
```

Algorithm 2 Reorganization Protocol for each node p

```
1.01 upon receipt of <MOVE,fs> from q do
1.02       if fs.val = p.val
1.03       then  //I send to myself that a fusion is needed.
1.04             send <MERGE,fs> to p
1.05       elseif p.val = prefix(fs.val)
1.06       then  if ∃s ∈ p.sons| s.val = prefix(fs.val)
1.07             then  // fs is in the subtrie of s, Case (a) in Figure 1
1.08                   send <MOVE,fs> to s;
1.09             elseif ∃s ∈ p.sons| fs.val = prefix(s.val)
1.10             then  // s is in the subtrie of fs, Case (b) in Figure 1
1.11                   p.sons := p.sons ∪ {fs}; p.sons := p.sons \ {s};
1.12                   send <MOVE,s> to fs;
1.13             elseif ∃s ∈ p.sons | p.val < PGCP(s.val, fs.val)
1.14             then  // fs and s have a PGCP which is greater than p.val
1.15                   // Case (c) in Figure 1
1.16                   Newnode(PGCP(fs.val, s.val), s, fs); p.sons := p.sons \ {s};
1.17             else  // fs is one of my sons, Case (d) in Figure 1
1.18                   p.sons := p.sons ∪ {fs};
1.19             endif
1.20       else  if p.father ≠⊥
1.21             then  send <MOVE,fs> to p.father
1.22             else  if fs.val = prefix(p.val)
1.23                   then  // I am in the subtrie of fs
1.24                         send <MOVE,p> to fs;
1.25                   else  // p and fs are brothers
1.26                         p.sons := p.sons ∪ Newnode(PGCP(fs.val, f.val), fs, p);
1.27                   endif
1.28             endif
1.29       endif

2.01 upon receipt of <MERGE,fs> from q do
2.02       Gluing(q);
2.03       Sorting of p.sons in the lexicographic order in Table t_s;
2.04       for  i = 0 to  t_s.length() do
2.05             if t_s[i].val = t_s[i + 1].val
2.06             then  send <MERGE,t_s[i + 1]> to t_s[i];
2.07                   i := i + 1;
2.08             elseif t_s[i].val = prefix(t_s[i + 1].val)
2.09             then  send <MOVE,t_s[i + 1]> to t_s[i];
2.10                   p.sons := p.sons \ {t_s[i + 1]};
2.11                   i := i + 1
2.12             elseif p.val < PGCP(t_s[i].val, t_s[i + 1].val)
2.13             then  p.sons := p.sons ∪ Newnode(PGCP(t_s[i].val, t_s[i + 1].val),
2.14                   t_s[i], t_s[i + 1]);
2.15                   p.sons; = p.sons \ {t_s[i], t_s[i + 1]};
2.16                   i := i + 1;
2.17             endif
2.18       done
```

1. The value of p is a prefix of fs's value—Line 1.05. In that case, following the four cases described in Figure 1, fs is eventually placed at the right place in the subtrie of p—refer to Lines 1.06 to 1.19. The resulting trie is a $PGCP$ tree.

2. The value of p is not a prefix of fs. Again, there are two cases to consider:

   (a) Node p has no father ($p.father = \perp$)—Line 1.22 to 1.28. In that case, if $fs.val$ is a prefix of p, then p (and its subtrie) becomes the node to be placed in fs—Line 1.24. Otherwise, p and fs become the two sons of a new root node q such that $q.val = PGCP(p, fs)$—Line 1.26. The trie is then clearly a $PGCP$ tree.

   (b) Node p has a father. Then, fs is moved to the father of p—Line 1.21. By induction of the above discussion, either fs eventually moves on a node q such that $q.val = prefix(fs.val)$ or fs eventually reaches the root of the trie. The former case is equivalent to Case 1, the latter to Case 2a.

If p and fs merge, then there are four cases to consider after p and fs glued together into p:

1. There exists a pair of sons s_i, s_j of p such that $s_i.val$ is a prefix of $s_j.val$. Then, s_j is moved toward s_i—Lines 2.08 to 2.11. This case is similar to the above Case 1 (Cases (a) or (b) in Figure 1).

2. There exists a pair of sons s_i, s_j of p such that $PGCP(s_i, s_j) > p.val$. Then, s_i and s_j become the two sons of a new son q of p such that $q.val = PGCP(p, fs)$—Lines 2.12 to 2.16. This case is also similar to the above Case 1 (Case (c) in Figure 1).

3. There exists a pair of sons s_i, s_j of p such that $s_i.val = s_j.val$. This case is solved by initiating a recursive merging between s_i and s_j—Lines 2.05 to 2.07. This case is solved by induction on s_i and s_j.

4. There exists no pair of sons s_i, s_j of p satisfying either Case 1, 2, or 3. In that case, the subtrie of p clearly satisfies the properties of a $PGCP$ tree.

□

From Lemmas 2 and 3 follows:

Theorem 1 *Under Assumption 1, Algorithm 1 and Algorithm 2 provide a PGCP tree reconstruction after the crash of a physical node.*

5 Conclusion and Future Work

In this paper, we have presented a fault-tolerant protocol in case of node crashes in a Proper Common Greatest Prefix tree search system. This protocol can be coupled with a replication strategy to lower the costs related to high replication factors. This protocol allows the reconnection and repair of subtries after the crash of one or more nodes. This algorithm guarantees to recover a consistent PGCP tree after a finite time and thus to avoid partially replication.

Our future work will consist in connecting the two mechanisms (replication and repair) in order to minimize the cost of fault-tolerance on dynamic platforms. We will also develop and validate experimentally the mechanisms exposed in this paper on the Grid'5000 platform of the french ministry of research. The aim of such experimentation will be to see the performance

of the repair algorithm and to see its capacity to answer clients' requests facing different levels of dynamicity. Moreover, we will be able to see starting from which level of dynamicity the repair mechanism is no more efficient alone, and then how we can progressively inject some replication as the dynamicity level increases.

References

[1] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Peer-to-Peer Computing*, pages 33–40, 2002.

[2] J. Aspnes and G. Shah. Skip Graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–393, January 2003.

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of Pervasive 2002*, 2002.

[4] S. Basu, S. Banerjee, P. Sharma, and S. Lee. NodeWiz: Peer-to-Peer Resource Discovery for Grids. In *5th International Workshop on Global and Peer-to-Peer Computing (GP2PC) in conjunction with CCGrid, May 2005*, 2005.

[5] E. Caron, F. Desprez, and C. Tedeschi. A dynamic prefix tree for the service discovery within large scale grids. In IEEE, editor, *The Sixth IEEE International Conference on Peer-to-Peer Computing, P2P2006*, Cambridge, UK., September 6-8 2006.

[6] E.J.H. Chang. Echo Algorithms: Depth Parallel Operations on General Graphs. *IEEE Trans. on Software Engineering*, SE-8:391–401, 1982.

[7] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range Queries in Trie-Structured Overlays. In *The Fifth IEEE International Conference on Peer-to-Peer Computing*, 2005.

[8] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *IPTPS'03*, pages 118–128, 2003.

[9] Gnutella. http://www.gnutella.com.

[10] KaZaA 2005. The KaZaA Web Site. http://www.kazaa.com.

[11] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, December 2004.

[12] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Prefix Hash Tree An indexing Data Structure over Distributed Hash Tables. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, St. John's, Newfoundland, Canada, July 2004.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Adressable Network. In *ACM SIGCOMM*, 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-To-Peer Systems. In *International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[15] C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems. *IEEE Internet Computing*, 8(3):19–26, 2004.

[16] A. Segall. Distributed Network Protocols. *IEEE Transactions on Information Theory*, IT-29:23–35, 1983.

[17] Y. Shu, B.-C. Ooi, K.-L. Tan, and A. Zhou. Supporting Multi-Dimensional Range Queries in Peer-to-Peer Systems. In *Peer-to-Peer Computing*, pages 173–180, 2005.

[18] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.

[19] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *DBISP2P*, 2003.

[20] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.